UNITED STATES PATENT APPLICATION

SYSTEM AND METHOD FOR PROCESSING MEMORY INSTRUCTIONS

INVENTORS:

Gregory J. Faanes
Eric P. Lundberg
Steven Scott
Robert Baird

Schwegman, Lundberg, Woessner, & Kluth, P.A.
1600 TCF Tower
121 South Eighth Street
Minneapolis, Minnesota 55402
ATTORNEY DOCKET 1376.711US1

SYSTEM AND METHOD FOR PROCESSING MEMORY INSTRUCTIONS

Related Applications

{INSERT OTHER APPLICABLE SV2 APPLICATIONS.}

Field of the Invention

The present invention relates generally to the field of multi-processor systems, and more particularly to processing memory instructions in a multi-processor system.

Background of the Invention

Multi-processor computer systems include a number of processing nodes connected together by an interconnection network. Typically, each processing node includes one or more processors, a local memory, and an interface circuit connecting the node to the interconnection network. The interconnection network is used for transmitting packets of information between processing nodes.

Distributed, shared-memory multiprocessor systems include a number of processing nodes that share a distributed memory element. By increasing the number of processing nodes, or the number of processors within each node, such systems can often be scaled to handle increased demand. In such a system, the processors may include one or more scalar processing units. These scalar processing units help control loading data from, and storing data to, addressable memory space in the distributed-memory system.

In order to load and store data, scalar processing units need to identify the proper address space for the data. In the past, individual nodes often have not had efficient or robust address identification mechanisms. In addition, such nodes often have not been able to handle multiple memory requests effectively (when scaling to large system size) or interface well with local cache requests and allocation.

Therefore, there is a need for a processing unit that addresses these and other shortcomings.

Summary of the Invention

To address these and other needs, various embodiments of the present invention are provided. One embodiment of the invention provides a method for processing a memory instruction. In this embodiment, the method includes obtaining a memory instruction, obtaining one or more memory address operands, creating a virtual memory address using the one or more memory address operands, translating the virtual memory address into a physical memory address, and executing the memory instruction on a cache controller, wherein the cache controller uses the memory instruction and the physical memory address to determine whether to access a portion of a local or a remote cache. In some embodiments, the obtaining of the memory instruction includes obtaining a memory instruction selected from a group consisting of a scalar load instruction, a scalar store instruction, a prefetch instruction, a synchronization instruction, and an atomic memory operation (AMO) instruction.

Another embodiment provides a computerized method that includes obtaining a memory request, storing the memory request in a first memory request container, and processing the memory request from the first memory request container by a cache controller. The processing of the memory request from the first memory request container by the cache controller includes identifying a type of the memory request, processing the memory request in a local cache as a function of a first condition, and processing the memory request using a second memory request container as a function of a second condition, wherein the processing of the memory request using the second memory request container includes updating a portion of the local cache with a portion of a remote cache.

These and other embodiments will be described in the detailed description below.

Brief Description of the Drawings

FIG. 1A illustrates a block diagram of a multi-streaming processor, according to one embodiment of the present invention.

FIG. 1B illustrates a block diagram of a node that includes four multi-streaming processors, according to one embodiment of the present invention.

FIG. 2A illustrates a format for a physical memory address, according to one embodiment of the present invention.

FIG. 2B illustrates a more detailed map of the physical memory address format shown in FIG. 2A, according to one embodiment of the present invention.

FIG. 3 illustrates a format for a virtual memory address, according to one embodiment of the present invention.

FIG. 4A illustrates a block diagram of a P chip having a scalar load store unit, according to one embodiment of the present invention.

FIG. 4B illustrates a more detailed block diagram of the scalar load store unit, according to one embodiment of the present invention.

FIG. 4C illustrates a block diagram of a cache controller in the scalar load store unit, according to one embodiment of the present invention.

FIG. 4D illustrates a first portion of a table that provides processing details of an Initial Request Queue (IRQ) in the scalar load store unit, according to one embodiment of the present invention.

FIG. 4E illustrates a second portion of the table that provides processing details of an Initial Request Queue (IRQ) in the scalar load store unit, according to one embodiment of the present invention.

Detailed Description

A novel system and method for processing memory instructions are described herein. In the following detailed description of the embodiments, reference is made to the accompanying drawings which form a part hereof, and in which are shown by way of illustration specific embodiments in which the invention may be practiced. These embodiments are described in sufficient detail to enable those who are skilled in the art to practice the invention, and it is to be understood that other embodiments may be utilized and that structural, logical and electrical changes may be made without departing from the scope of the present inventions. It is also to be understood that the various embodiments of the invention, although different, are not necessarily mutually exclusive. For example, a particular feature, structure or characteristic described in one embodiment may be included within other embodiments. The following description is, therefore, not to be taken in a limiting sense.

FIG. 1A and FIG. 1B illustrate specific hardware environments in which various embodiments of the present invention may be practiced. In one embodiment, the hardware environment is included within the Cray X1 System Architecture, which represents the convergence of the Cray T3E and the traditional Cray parallel vector processors. The X1 is a highly scalable, cache coherent, shared-memory multiprocessor that uses powerful vector processors as its building blocks, and implements a modernized vector instruction set.

FIG. 1A illustrates a block diagram of a multi-streaming processor (MSP), according to one embodiment. In this embodiment, MSP 100 is an eight-chip multi-chip module having four processor chips (P chips) and four custom cache chips (E chips). Each P chip contains a superscalar processor with a two-pipe vector unit, and the E chips implement a 2MB cache (0.5 MB each) that is shared by the P chips. Each P chip contains a small Dcache that is used for scalar references only, and the E chips are shared by all P chips for scalar data, vector data and instructions.

Each P chip has many features. The scalar and vector units support 32- and 64-bit operations on both integer and floating-point data. Most all computational functions, including the register files, are built using full custom logic for maximum speed and reduced area and power. The remaining logic is implemented in a standard-cell form (which is faster and more dense than a gate-array implementation). The scalar engine is 2-way dispatch, out-of-order issue, in-order completion with 2-deep branch prediction. There are 64 A registers and 64 S registers. The A registers are generally used to hold addresses and address/integer calculations. The S registers hold the results of either integer or floating-point calculations. S register values can also be used in vector/scalar operations (where a single value is used in conjunction with the multiple values of vector registers). The scalar engine has a 16KB instruction cache and a 16KB write through data cache. There are 32 vector registers of 64 elements each, implemented in two vector pipelines. (Thus the depth of each pipe is 32 elements.) Most 32-bit vector operations take place at twice the rate as for 64-bit operations. Multiple mask registers are implemented which will enable more codes to support higher levels of vectorization. Special instructions and hardware support are added to enable the four

P chips to cooperate on single job streams. Both 32-bit and 64 bit word sizes are supported. The A registers implement 32 and 64 bit integer functions while the S registers support the same integer functions and additionally support 32-bit and 64-bit floating point arithmetic. The vector pipes support 32- and 64-bit integer operations and 32- and 64-bit floating point operations. Each E chip has 0.5 MB of cache, tag support, coherency directories, extensive data routing, and support for synchronization. The E chips and P chips are IBM CMOS ASIC's.

Each scalar processor of a P chip delivers a peak of 0.8 Giga-instructions per second and 800 MFlops at a 400 MHz clock rate. The two vector pipes in each P chip provide 3.2 GFlops, with a logic clock (Lclk) of 400 MHz and a custom functional unit clock (Sclk) of 800 MHz. The links between the vector processors and the E chips are single-ended, 400 Mbaud, and include a 64-bit outgoing address/store data path, and a 64-bit incoming load data path plus miscellaneous control wires. Each E chip has four system ports to the local memory and network.

These links are differential, 800 Mbaud, and include 16 data bits wide in each direction plus miscellaneous control wires.

Each scalar processor has a 16 KB Dcache. The Dcache functions in write-through mode, and is kept coherent with the E chips (Ecache) through selective invalidations performed by the E chips. The scalar processors and the vector units share E chips. The E chips are responsible for enforcing ordering of memory references between processors as dictated by local memory synchronization instructions.

The scalar processors of P chips have many attributes, including nominal 400 MHz operation outside the custom logic. This includes instruction dispatch and issue. The scalar processors have two-way, superscalar dispatch (thus 0.8 Giga-operations per sec), and can issue up to two S register, two A register, 1 Load/Store, 1 Branch or 1 Vector instruction each clock. The scalar unit and V register pipelines each have three functional unit groups that are independent and are pipelined. Each of the functional unit groups can perform two operations per clock, thus giving twice the result rate as with single functional units. The functional units are: 1- add, logical, compares; 2- multiply, shift; 3- divide, logical, converts, specials (POP, BMM, etc.). Some integer functions are executed only from the S registers. The A and S registers support integer divides. The S registers also support floating point divide. (Floating divide only occurs in vectors.) The scalar processors provide 64 logical and 512 physical A and S registers. Register shadowing is used rather than general register renaming. (Each logical A and S register has 8 shadow registers). Branch prediction and speculative execution are also provided. Scalar instructions can execute speculatively, while vector instructions do not. The scalar processors have on-chip instruction caches of 16 KBytes, which are 2-way set-associative. 32-byte cache lines are used. The scalar processors also have on-chip data caches of 16 KBytes. These caches are used in a write-through mode. Load instructions can bypass the cache (no allocate). Cache lines are also 32 bytes. There are multiple

Sync instructions useful on both a local basis and global basis (across all MSP's in a system).

The vector processors in the P chips each include 32 vector registers, each of 64 elements. There are 64 elements in a vector, both in 32- and 64-bit modes. Both floating-point operations and integer/logical/ shift functions are provided. A two-pipe vector unit is provided in a vector processor, so that each pipe has 32 elements of all vector registers. Operation is overlapped and chained. The vector processors have custom circuits running at 800 MHz. These are used to speed up the vector register/mux/functional unit pipelines. Thus, the total vector flops are 3.2 GFlops per P chip, and 12.8 GFlops for the MSP. Vectors are decoupled from the scalar unit to allow the scalar unit to run ahead. The vector processors provide full support for 32-bit operands. 32-bit computations execute twice as fast as 64-bit. Stride-1 memory operations (the most common case) using 32-bit data are also executed at twice the rate. 32-bit operands will be stored two to a 64-bit vector element, but the operands in an element might not be consecutive (probably n and n+2). The way in which 32-bit operands are supported is determined by the Cray SV2 implementation (for this embodiment), and is not visible at the instruction-set level. The vector pipes each have a functional unit for Mask operations, in addition to the units listed above for the scalar processor.

FIG. 1B illustrates a block diagram of a node that includes four multi-streaming processors (MSP's), according to one embodiment. In this embodiment, node 102 includes each MSP 100 in a four MSP system. Node 102 is contained on a single printed circuit board. The sixteen M chips on node 102 contain memory controllers, network interfaces and cache coherence directories with their associated protocol engines. The memory system is sliced across the 16 M chips, round robin by 32-byte cache lines. Each M chip supports one slice. Bits 5 and 6 of the physical address determine the E chip with a processor, and bits 7 and 8 further specify one of four M chips connected to each E chip.

Each M chip resides in one of sixteen independent address slices of the machine, and the interconnection network provides connectivity only between corresponding M chips on different nodes. All activity (cache, memory, network) relating to a line of memory stays within the corresponding slice. Each M chip controls a separate sector of a slice. Slices expand (get more memory in each) as nodes are added so the number of sectors in each slice is equal to the number of nodes in a system.

Total peak local memory bandwidth for one node is 204.8 GB/s, or 51.2 GB/s per MSP. As each MSP 100 needs a maximum bandwidth of about 45 GB/s, there is bandwidth to support network traffic and I/O without greatly impacting computational performance. Each M chip contains two network ports, each 1.6 GB/s peak per direction.

Node 102 also contains two I chip I/O controller ASICs. These connect to the M chips and provide four I/O ports of 1.2 GB/s bandwidth, full duplex, off node 102. Each I chip contains two ports, 400 MB/s full duplex connections to 8 of the local M chips (one I chip connects to the even M chips and the other connects to the odd M chips), and a 1.6 GB/s full duplex connection to the other I chip. The total I/O bandwidth per module is thus 4.8 GB/s full duplex.

The memory on node 102 is distributed across the set of 16 M chips. Each M chip directly controls the resources contained on two daughter boards so that there are thirty-two daughter boards on node 102. The memory chips in the daughter boards are Direct Rambus DRAM. These chips have 16 internal banks and have 18 data pins that each run, with a 400 MHz clock, at an 800 Mbaud rate. Each chip then has a 1.6 GB/s read/write data rate. Being 18 bits wide, additional parts to support ECC are not needed. Daughter cards contain 16 chips organized into 4 memory channels of 4 chips each. Each memory channel is independent. Channels have a peak data bandwidth of 1.6 GB/s, so that the card supports a bandwidth of 6.4 GB/s. With 16 banks in a memory chip, a channel has 64 memory banks. Daughter cards with 64 Mbit, 128 Mbit, 256 Mbit or 512 Mbit chips are supported.

The design also accommodates chip densities of 1 Gbit if and when they become available, assuming they fit in the design envelope (size, power, etc.). As the memory parts are 18 bits wide instead of 16 in order to support ECC, the chip's bit densities are actually 72, 144, 288, 576 and 1152 Mbits.

Memory ordering, when required, can be provided in the four MSP node and between MSPs of different nodes by using one of the Sync instructions. In one embodiment, there are three different Sync instructions: Lsync instructions, Msync instructions, and Gsync instructions.

Lsyncs provide ordering among vector and scalar references by the same processor. The Lsync_s_v instruction guarantees that previous scalar references complete before subsequent vector references. The Lsync_v_s instruction guarantees that previous vector references complete before subsequent scalar references. Finally, the Lsync_v_v instruction guarantees that previous vector references complete before subsequent vector references. There are also a few other varieties that provide even lighter-weight guarantees. These instructions are used when the compiler or assembly language programmer either knows of a data dependence, or cannot rule out the possibility of a data dependence between the various classes of memory references.

Msyncs provide ordering among memory references by the processors of an MSP. The Msync instruction is executed independently on multiple processors within the MSP (typically by either all processors, or a pair of processors performing a "producer/consumer" synchronization). The Msync instruction includes a mask indicating which processors are participating, and all participating processors should use the same mask, or else hardware will detect the inconsistency and cause an exception. The regular Msync orders all previous references by all participating processors within the MSP before all subsequent references by all participating processors. It is essentially a memory barrier and control barrier wrapped into one. The vector Msync acts the same, but only applies to vector references. Lastly, the producer Msync is intended to situations in which one processor is producing results for one or more other processors to consume. It doesn't require the producer to wait to see earlier results possibly written by the "consumers." Msyncs are highly optimized in this embodiment. Vector store addresses are sent out to the E chips long before the actual data is available; the store data are sent along later. Load requests that occur after an Msync are checked against the earlier store addresses. If there is no match, the loads are serviced, even before the data for stores occurring before the Msync have been sent to the E chip.

Gsyncs provide ordering among references made by multiple MSPs. They are generally used whenever data is shared (or potentially shared) between MSPs. Like Msyncs, Gsyncs include a mask of participating processors within an MSP, and all participating processors should issue a Gsync with a consistent mask. The regular Gsync prevents any subsequent memory references by the participating processors from occurring until all previous loads have completed and all previous stores have become globally visible. A Gsync should be used, for example, before performing a synchronization operation (such as releasing a lock) that informs other processors that they can now read this processor's earlier stores. Several variants of Gsync are provided, including versions optimized for lock acquire and lock release events.

FIG. 2A illustrates a format for a physical memory address, according to one embodiment. In this embodiment, a 46-bit (64 TBytes) physical memory address is supported. The node size for this embodiment is a board containing four MSP's and 16 M chips. Physical memory address format 200 contains bits 47..0. Bits 35..0 represent an offset (into memory). Bits 45..36 represent the node. Bits 47..46 represent the physical address space. The physical memory format allows for up to 1024 nodes (4096 MSP's) and 64 GBytes of physical memory per node. Physical pages are allocated on a per-node basis. That is, any given physical page is distributed uniformly across the 16 sectors (the memory controlled by a given M chip) of a single node. This embodiment provides three parallel, physical address spaces, which are selected by two extra bits at the top of the physical address.

FIG. 2B illustrates a more detailed map of the physical memory address format shown in FIG. 2A, in one embodiment. The mapping of a physical address to a destination location is dependent on the hardware implementation (as opposed to being specified in the user-level architecture). Address mapping should be done so that parallelism can be easily exploited by having the map such that multiple transactions can be requested and satisfied simultaneously with minimum hardware complexity. Bits 4..0 represent the byte in the line. Bits 6..5 represent the quadrant (E chip). Bits 8..5 collectively represent the slice/section (M chip). Bits 11..9 represent the memory channel. Bits 13..12 represent the memory chip for the memory channel, and bits 17..14 represent the bank for the memory chip. Bits 35..18 represent the memory chip address, and bits 45..36 represent the node number (in the system). Bits 47..46 represent the address space. Memory size options and configuration changes (including memory degrades) can modify this map. The map supports memory chips up to 1 Gbit density. There are three address spaces: coherent main memory, memory-mapped register space, and I/O device space. Coherent main memory may be cached.

FIG. 3 illustrates a format for a virtual memory address, according to one embodiment. In this embodiment, virtual memory address format 300 contains a 64-bit virtual address space. Bits 37..0 represent a virtual offset into virtual memory space, wherein potential page boundaries range from 64 KB to 4 GB. Bits 47..38 represent the VNode (i.e., virtual node). This is used by the hardware when performing remote address translation. Bits 61..48 should be set to zero in this implementation. Bits 63..62 specify the memory region, which determines the type of address translation used in kernel mode. The virtual address space can be considered a flat virtual address space for uniprocessor, or symmetric multiprocessing applications. As stated, this embodiment supports eight page sizes ranging from 64 KB to 4 GB. Thus, the page boundary can vary, from between bits 15 and 16, to between bits 31 and 32.

In various embodiments of the invention, virtual addresses used for instruction fetches and data references are first translated into physical addresses before memory is accessed. These embodiments support two forms of address translation: source translation, and remote translation. The first form of address translation is source translation, in which a virtual address is fully translated by a Translation Look-aside Buffer (TLB) on a local P chip to a physical address on an arbitrary node. The second form of address translation is remote translation, in which the physical node number is determined by a simple translation of the virtual address VNode field, and the remaining virtual address VOffset field is sent to the remote node to be translated into a physical address offset via a Remote-Translation Table (RTT). The type of address translation performed is based upon values in the TLBControl control register and the virtual address itself. Remote translation is performed if all of the following three conditions are true: (1) Remote translation is enabled (TLBcontrol.RemoteTrans = 1); (2) The virtual address is to the used region (Bits 63..62 = 00 in the virtual address); and (3) The virtual address references a remote node (Bits 47..38 in the virtual address are not equal to TLBcontrol.MyNode). If any of the above conditions are false, then source translation is performed. Remote translation can be enabled/disabled on a per-processor basis.

FIG. 4A illustrates a block diagram of a P chip having a scalar load store unit, according to one embodiment of the present invention. In this embodiment, the P Chip contains three sections. One is Scalar Section (SS) 400. SS 400 is an out-of-order, two-way issue superscalar processor and contains the Instruction Fetch Unit (IFU), the Dispatch Unit (DU), the AS Unit (ASU), the Load/Store Unit (LSU) 406, and the Control Unit (CU). Another section is Vector Section (VS) 402. VS 402 contains the Vector Dispatch Unit (VDU), the Vector Unit (VU) and the Vector Load/Store Unit (VLSU). This section contains a two-pipe vector processor capable of executing eight floating-point operations and four memory operations per Lclk. The final section is Memory Interface Section (MS) 404. MS

404 contains the Ecache Interface Unit (EIU), which communicates with the E Chips to ensure high bandwidth between the SS/VS and the external cache. The P Chip uses three copies of a Custom Block (CB) in two different units. The ASU and VU use a common custom block for its large register file and fast functional units. The ASU uses one copy of the CB while the VU uses two copies, one for each vector pipe.

In this embodiment, SS 400 is a high performance superscalar processor. It dispatches, in-order, up to two instructions per Lclk, executes instructions out-of-order within the various units, and then graduates in-order up to two instructions per Lclk. SS 400 also implements speculative execution, register renaming, and branch prediction to allow greater out-of-order execution. SS 400 can predict up to two branch instruction, and uses a Branch History Table (BHT), a Jump Target Buffer (JTB), and Jump Return Stack (JRS) to help insure a high branch prediction rate. SS 400 also contains two 64-bit wide register files, A Registers (AR) and S Registers (SR). The AR's are used mainly for address generation. There are 64 logical AR's and 512 physical AR's that are renamed. The SR's are used for both integer and floating-point operations. There are 64 logical SR's and 512 renamed physical SR's.

SS 400 is capable of issuing up to two integer operations per Lclk using the AR's and up to one SR instructions per Lclk that can be either integer or floating point. The decoupled LSU of SS 400 can issue, in order, one load or store per Lclk, which may then execute out-of-order with respect to previous scalar memory operations. SS 400 is also able to issue one branch instruction per Lclk, which allows one branch prediction to be resolved per Lclk. SS 400 includes separate first level of caches for instructions and scalar data.

The Instruction Cache (Icache) is 16KBytes in size and is two-way set associative. Each Icache line is 32 Bytes. The Data Cache (Dcache) is also 16KBytes and two-way set associative with a 32 Byte line size. The Icache is virtually indexed and virtually tagged while the Dcache is virtually indexed and physically tagged. The Dcache is write through.

All instructions flow through the P Chip's units by first being fetched by the IFU, and then sent to the DU for distribution. In the DU, instructions are decoded, renamed, and entered into the Active List (AL).

Instructions can enter the AL in a speculative state. Speculative instructions may execute, but their execution cannot cause permanent processor state changes while the instructions remain speculative. Instructions in the AL proceed from speculative, to scalar committed, to committed and then to graduated. Scalar committed and committed instructions are not branch speculative. After commitment, an instruction proceeds to graduated and is removed from the AL. Instructions cannot be marked complete until the instruction can be removed from the AL, and at least requires that all trap conditions are known, all scalar operands are read, and any scalar result is written.

Scalar instructions are dispatched by the DU in program order to the AU and/or the SU. Most scalar instructions in the AU and SU are issued out-of-order, read the AR or SR, execute the indicated operations, write the AR or SR and send instruction completion notice back to the DU. The DU then marks the instruction complete and can graduate the scalar instruction when it is the oldest instruction in the AL. All scalar memory instructions are dispatched to the AU. The AU issues the memory instructions in-order with respect to other scalar memory instructions, reads address operands from AR and sends the instruction and operands to LSU 406. For scalar store operations, the memory instruction is also dispatched to the AU or SU to read the write data (to be stored) from the AR or SR and send this data to LSU 406.

LSU 406 performs address translation for the memory operations received from the AU in-order, sends instruction commitment notice back to the DU, and executes independent memory operations out-of-order. For scalar loads, when load data is written into the AR or SR, the AU or SU transmits instruction completion notice back to the DU. Scalar store instruction completion is sent by the Ecache

Interface Unit (EIU) to the DU when the write data has been sent off to the E chip.

Branch instructions are predicted in the IFU before being sent to the DU. The DU dispatches the branch instruction to either the AU or SU. The AU or SU issues the instruction, reads AR or SR, and sends the operand back to the IFU. The IFU determines the actual branch outcome, signals promote or kill to the other units and sends completion notice back to the DU.

Vector instructions are dispatched in-order from the DU to the VDU. The VDU dispatches vector instructions to both the VU and VLSU in two steps. First, all vector instructions are vpredispatched in-order in the VDU after all previous instructions are scalar committed. The VDU separates the stream of vector instructions into two groups of vector instructions, the VU instructions and VLSU instructions. All vector instructions are sent to the VU, but only vector memory instructions and instructions that write VL and VM sent to the VLSU.

FIG. 4B illustrates a more detailed block diagram of the scalar load store unit, according to one embodiment of the present invention. In this embodiment, LSU 406 includes an address generator and four cache controllers (408). Each cache controller 408 contains an interface to one of the four Ecache ports, and the portion of the Dcache associated with that port.

LSU 406 processes scalar loads, scalar stores, prefetches, syncs, and atomic memory operations. Instructions are received from the A Unit (AU), and can be processed when the address operands (Aj and optionally Ak) have arrived. The address add is performed for prefetches and all aligned loads and stores to generate a virtual address. Atomic Memory Operations (AMO's) and unaligned loads and stores use Aj directly as an address. All instructions that reference memory (all but the syncs) are then translated as needed (using either source or remote translation, as described earlier), and checked for address errors. Accesses to the kphys memory region are not translated. The TLB reports completion of translation and any errors back to the Active List.

After the TLB, instructions are placed into the Initial Request Queue (IRQ). The IRQ is a container, and contains 8 entries, allowing up to 8 scalar references to be translated after an "Lsync V,S" instruction before the Lsync completes. While these scalar references can't access the Dcache until the Lsync completes, allowing them to pass translation can permit subsequent vector instructions to be dispatched, which will improve performance for certain loops.

From the IRQ, instructions are sent to one of the four Cache Controllers 408 (CC0 - CC3), steered by physical address bits 6..5. Sync instructions are broadcast to each of the four cache controllers 408. The Flow Info Queue maintains steering information for scalar store and AMO data values. Each entry records the type of operand(s) and the port to which they should be steered. Data values arrive in order from each of the AU and SU, and are steered accordingly. Ak values are used only for AMO's with two operands.

FIG. 4C illustrates a block diagram of a cache controller in the scalar load store unit, according to one embodiment of the present invention. Controller 408 contains the tag, state and data arrays for that portion of the Dcache corresponding to Ecache port 'x' (i.e.: having physical address bits 6..5 = 'x'). It includes two primary request-processing pipelines: the IRQ, from the address generation logic, and the Forced Order Queue (FOQ), through which requests are routed when they cannot be serviced immediately from the IRQ. Both the IRQ and FOQ are containers that are able to both hold and service requests.

Physical address bits 12..7 of a request are used as the index into the local Dcache (which contains 64 indices times two ways, for a total of 128 cache lines). A request from the IRQ simultaneously indexes into the Dcache tag, state and data arrays, and also performs an associative, partial address match with all entries in the FOQ. The indices and control information for all FOQ entries are replicated for this purpose in the FOQ index array. No request is allowed to leave the IRQ until it is not branch speculative.

Read requests that hit in the Dcache and have no potential matches in the FOQ are serviced immediately. Read and write requests that miss in the Dcache and have no potential matches in the FOQ cause the line to be allocated in the Dcache. A request packet is sent to the Ecache immediately, and the instruction is placed into the FOQ to await the response from the E chip. In the case of a Dcache allocation, the state of the line is immediately changed to valid; there is no "pending" state. The simple presence of a request for a given line in the FOQ serves the same purpose as a pending state. A subsequent request to the line that is processed from the IRQ before the newly-allocated line has arrived back from the E chip will detect the matching request in the FOQ and will thus not be satisfied from the Dcache. If a new request from the IRQ matches the partial address of any valid entry in the FOQ, then there is a potential exact address match with that entry, and the request is routed through the FOQ to maintain proper request ordering. All AMO's and I/O widget space references are sent through the FOQ, as they do not allocate Dcache lines and can never be serviced from the Dcache. Requests that miss in the Dcache and would otherwise have allocated a Dcache line do not do so when they match an FOQ entry; they are run through the FOQ and passed on to the Ecache. Simplifying the handling of this relatively infrequent event significantly simplifies LSU 406.

The FOQ is logically two separate queues unified in a single structure: one queue for accesses to the Dcache, and one queue for accesses to the Ecache. Each entry in the FOQ can be marked as accessing the Dcache and/or accessing the Ecache. FIFO ordering within each class is preserved. That is, all Dcache requests are kept in order with respect to each other, and all Ecache requests are kept in order with respect to each other. However, Dcache-only requests and Ecache-only requests may be dequeued in a different order than they were enqueued.

An FOQ entry that is marked to access both the Ecache and Dcache may logically be dequeued from the Ecache queue before being dequeued from the Dcache queue. After doing so, the request will still remain in the FOQ, but be marked only as a Dcache request. This might happen, for example, for a write request which is able to send its write through to the Ecache, but not yet able to write to the Dcache because a newly allocated Dcache line has not yet returned from memory. In general, the Ecache queue will "run ahead" of the Dcache queue, as the head of the Dcache queue will often be waiting for a response from the E chip, whereas requests to the E chip can generally be sent as soon as they are ready.

Sync instructions are always sent through the FOQ. A marker for an "Lsync S,V" is simply passed on to the E chip port after all Ecache requests in front of it have been sent. This marker informs the E chip port arbiter that previous scalar references from this port have been sent to E. A Gsync instruction marker is similarly passed through the FOQ and sent to the E chip port arbiter after all previous scalar references have been sent.

Processing an "Lsync V,S" instruction from the IRQ causes the head of the IRQ to block, preventing subsequent scalar references from accessing the Dcache or being sent to the Ecache until all vector references have been sent to the Ecache and all vector writes have caused any necessary invalidations of the Dcache. Vector write invalidations are performed using a separate port to the Dcache tags. Once all vector writes before the Sync have been run through the Dcache, LSU 406 is signalled to unblock the IRQ's at each cache controller 408. In the meantime, each cache controller 408 sends its "Lsync V,S" marker through its FOQ and on to the E chip port arbiter (in one implementation).

Markers for Msync instructions are also sent through the FOQ and passed to the port arbiter. Processing a regular Msync instruction from the IRQ causes cache controller 408 to go into "Dcache bypass" mode. In this mode, reads and writes are forced to miss in the Dcache and are sent to the Ecache following the Msync marker. This causes them to see the results of memory references made before the Msync by other P chips participating in the Msync. Once an E chip port arbiter has received the Msync marker from scalar LSU 406 and the vector LSU, it sends an Msync marker to the associated E chip. Once all participating P chips have sent their Msync markers to an E chip and E has sent the Dcaches any invalidations from writes before the Msync, the E chip sends Msync completion markers back to the P chips. An Msync completion marker from a given E chip turns off "Dcache bypass" mode at the associated cache controller 408. External invalidations received from the Ecache are performed using a separate port to the Dcache tags.

While in bypass mode, read requests that would otherwise hit in the Dcache actually cause the corresponding Dcache line to be invalidated. This is done so that a subsequent read that is processed just after Dcache bypass has been turned off will not read a line out of the Dcache, effectively reading earlier data than an earlier read request which is heading out to the Ecache via the FOQ. The "Dcache bypass" mode can also be permanently turned on via the DiagConfig control register.

FIG. 4D and 4E illustrate a first and second portion of a table that provides processing details of an Initial Request Queue (IRQ) in the scalar load store unit, according to one embodiment of the present invention. Table 410 provides a complete listing of actions taken by a tag/state engine when processing IRQ requests. The first four columns indicate whether a cache controller 408 is currently in "Dcache bypass" mode, the request type, whether the request hits in the Dcache and whether the request matches the partial address in any FOQ entry. The remaining columns indicate the action that is taken to process the request.

In some cases, a message is sent directly to the E chips (via the port arbiter) by the IRQ tag/state engine. In two cases, the request is serviced directly from the Dcache and in all others it is placed into the FOQ. The "E", "D" and "P" columns of table 410 indicate whether an FOQ entry is marked as accessing the Ecache, accessing the Dcache, and/or pending, respectively. An entry marked pending will always be marked as a Dcache entry as well. It cannot be dequeued from the head of the Dcache queue until the matching response arrives from the Ecache, clearing its pending status.

The "Allocate" column indicates if a request causes a (least-recently-used) allocation in the Dcache, or causes an invalidation of the matching line. The "ORB Entry" indicates whether the request allocates an entry in the Outstanding Request Buffer (ORB). All requests that will receive a response from an E chip allocate an ORB entry. The ORB contains 16 entries, and is indexed by an transaction ID (TID) sent to the Ecache in the request packet and returned in the response. Each ORB entry contains a request type, load buffer index, an A or S register number, Dcache index and way, FOQ index, s/dword flag, and physical address bits 4..2 of the request. The ORB entry specifies what to do with the response when it is received from the E chip. A "Read" entry causes the response to be placed into the Dcache and the requested word to be sent to a register. A "ReadUC" entry causes the result to be sent only to a register, and not written into the Dcache. A "Prefetch" entry causes the result to be written to the Dcache, but not returned to a register. All scalar requests that allocate an ORB entry expect either a single s/dword or a full cache line in return. For full cache line requests, the requested word will be returned first, with the rest of the cache line following in increasing address, modulo the cache line size. For both word and cache line requests, the requested s/dword is returned directly to the register from the E port via the load buffers. Thus, the FOQ entry for a load that caused a Dcache allocation is marked as a dummy. It exists so that subsequent IRQ requests to the same line will detect a match and not access the pending line before it returns from the E chip. When a dummy request is dequeued from the FOQ, it is simply discarded; it does not access the Dcache.

Ecache requests at the head of the FOQ can be dequeued as follows. Regular writes, I/O writes, AMO reads (which return data) and writes (which do not) are dequeued after they are committed and after their data is available. I/O reads and Syncs are dequeued after they are committed. ReadUC and ReadNA requests can be dequeued immediately.

Dcache requests at the head of the FOQ can be dequeued as follows. Dummy requests can be dequeued after their matching response from E has returned and written the Dcache. Reads can be dequeued immediately. Allocating writes (those that were marked pending) can be dequeued after their matching response from E has returned and written the Dcache, they are committed, and their store data is available. Other writes are dequeued after they are committed, and their store data is available.

Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that any arrangement that is calculated to achieve the same purpose may be substituted for the specific embodiment shown. This application is intended to cover any adaptations or variations of the described embodiments of the present invention.